1. Notationen und Algorithmen zu Rechenausdrücken

1.1. Einleitung

Die Mathematik kennt verschiedenen Schreibweisen für Rechenausdrücke (Terme) , die zu einander äquivalent sind:

- ### 1.1.1. Die Infix-Notation (Vorsilbe *"in"* für *innerhalb* bzw. *dazwischen*)

  Die **Infix-Notation** ist die vertraute Schreibweise der Schulmathematik.

  4 + 7

  Bei ihr wird jeder Operator (z.B. das Plus-Zeichen) **möglichst zwischen seinen Operanden** (z.B. Zahlen oder Variablen) **geschrieben**.

  Im Unterschied zu den anderen hier behandelten Schreibweisen benötigt die Infix-Notation Vorrangigkeitsregeln (z.B. Punkt- vor Strichrechnung) und Klammersetzungen, um Rechenausdrücke korrekt zu formulieren. Infix-Notation sehen damit schöner aus und sind für das Auge leichter zu lesen als andere Schreibweisen.

- ### 1.1.2. Die Präfix-Notation (Vorsilbe *"prä"* für *davor*)

  Die **Präfix-Notation** ist eine weniger bekannte Schreibweise der Mathematik.

  \+ 4 7

  Bei ihr wird jeder Operator (z.B. das Plus-Zeichen) **vor seinen Operanden** (z.B. Zahlen oder Variablen) **geschrieben**.

  Die Präfix-Notation wurde von dem polnischen Mathematiker Łukasiewicz eingeführt, und sie kommt ganz ohne Klammersetzungen und Vorrangigkeitsregeln aus. Für das Auge lesen sich Präfix-Notationen unangenehmer als Infix-Notationen, sind dafür aber bei technischen Umsetzungen (z.B. bei Programmierungen) leichter zu handhaben.

  Je nach Vorliebe von Fachauthoren wird die *Präfix-Notation* auch

  - *Łukasiewicz-Notation*,
  - ***Polnische Notation***,
  - ***Normale Polnischen Notation*** (abgekürzt: **NPN**),
  - *prefix notation* (auf englisch)

  genannt.

- ### 1.1.3. Die Postfix-Notation (Vorsilbe *"post"* für *dahinter*)

  Die **Postfix-Notation** ist ebenfalls eine weniger bekannte Schreibweise der Mathematik.

  4 7 +

  Bei ihr wird jeder Operator (z.B. das Plus-Zeichen) **hinter seinen Operanden** (z.B. Zahlen oder Variablen) **geschrieben**.

Die Schreibweise ist damit die Umkehrung der *Normalen Polnischen Notation* und wird von Fachauthoren auch

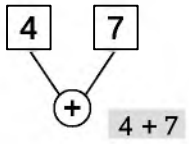
- *Umgekehrte Łukasiewicz-Notation*,
- *Reverse Łukasiewicz-Notation*,
- ***Umgekehrte Polnischen Notation*** (abgekürzt: ***UPN***),
- *Reverse Polnische Notation*,
- ***reverse polish notation*** (auf englisch, abgekürzt: ***RPN***)

genannt, und auch sie kommt ganz ohne Klammersetzungen und Vorrangigkeitsregeln aus.

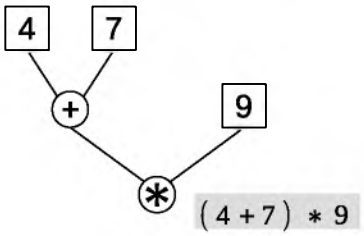
Für das Auge lesen sich Postfix-Notationen ebenfalls unangenehmer als Infix-Notationen, doch sind sie bei technischen Umsetzungen (z.B. bei Programmierungen) sehr leicht zu handhaben.

• 1.1.4. Termbäume (Rechenbäume)

Termbäume (Rechenbäume) sind eine wenig bekannte Strukturschreibweise der Mathematik. Mit ihnen können ganz einfache Rechenausdrücke (Terme) abgebildet werden, z.B.:



Es können aber auch längere Terme abgebildet werden, z.B.:



Mitunter können Termbäume sehr umfangreiche Abbildungen sein, und es braucht Zeit sie vollständig zu skizzieren und niederzuschreiben. Doch dafür kann ihr Ablauf intuitiv erfaßt und in allen Details nachvollzogen werden.

Es sei nochmals darauf hingewiesen, daß alle vier Schreibweisen äquivalent sind. Je nach dem, welche Anforderungen einer mathematischen Behandlung zu Grunde liegen, bieten diese vier Schreibweisen unterschiedliche Vor- und Nachteile; und anders als in der Schule ist die Infix-Notation dabei nicht immer die beste Wahl. Der Umgang mit diesen Schreibweisen sollte daher vertraut sein, um den jeweils bestmöglichen Nutzen daraus zu ziehen.

1.2. NPN und UPN näher kennenlernen

(NORMALE POLNISCHE NOTATION und *UMGEKEHRTE POLNISCHE NOTATION)*

Um beispielsweise den schulmathematische Rechenausdruck $a + b$ von Infix-Notation nach NPN umzuwandeln, nehmen wir das darin enthaltene Plus-Zeichen (das ist der Operator) heraus und notieren dieses vor die Operanden a und b .

Damit erhalten wir für NPN: $+ \; a \; b$

Notieren wird das Plus-Zeichen (den Operator) hingegen hinter die Operanden a und b, so wandeln wir die Infix-Notation nach UPN um.

Und wir erhalten für UPN: $a\ \ b\ \ +$

Übung:

Decke mit einem Blatt Papier die rechte und die mittlere Spalte in der nachstehenden Tabelle ab, und versuche anschließend die in der linken Tabellenspalte stehenden Infix-Ausdrücke der Schulmathematik in Präfix-Notation (NPN) und in die Postfix-Notation (UPN) umzuwandeln.

Gehe zeilenweise vor und notiere die jeweilige NPN- und UPN-Schreibweise. Vergleiche anschließend direkt, ob Deine Umwandlungsergebnisse richtig sind, oder wo Dein Denkfehler lag. Das abdeckende Papierblatt schiebst Du dazu tiefer, so daß Dir das korrekte Ergebnis für in der jeweiligen Zeile angezeigt wird.

Schulmathematik	NPN-Schreibweise	UPN-Schreibweise
$a + b$	$+\ a\ b$	$a\ b\ +$
$4 + 7$	$+\ 4\ 7$	$4\ 7\ +$
$c - d$	$-\ c\ d$	$c\ d\ -$
$9 - 3$	$-\ 9\ 3$	$9\ 3\ -$
$(a + b) * (c - d)$	$*\ +\ a\ b\ -\ c\ d$	$a\ b\ +\ c\ d\ -\ *$
$(4 + 7) * (9 - 3)$	$*\ +\ 4\ 7\ -\ 9\ 3$	$4\ 7\ +\ 9\ 3\ -\ *$
$(a + b) \div (c - d)$	$\div\ +\ a\ b\ -\ c\ d$	$a\ b\ +\ c\ d\ -\ \div$
$(4 + 7) \div (9 - 3)$	$\div\ +\ 4\ 7\ -\ 9\ 3$	$4\ 7\ +\ 9\ 3\ -\ \div$
$\sin(a)$	$\sin\ a$	$a\ \sin$
$\sin(62)$	$\sin\ 62$	$62\ \sin$
$\sin(r * \pi)$	$\sin\ *\ r\ \pi$	$r\ \pi\ *\ \sin$
$\sin(0{,}7 * \pi)$	$\sin\ *\ 0{,}7\ \pi$	$0{,}7\ \pi\ *\ \sin$

1.2.1. NPN- und UPN-Terme handhaben

NPN- und UPN-Rechenausdrücke werden durchgängig von links nach rechts abgearbeitet. Es gibt weder Vorrangigkeiten noch Klammersetzungen. Damit werden Rechenoperationen transparenter und sparen Schreibarbeit bei komplexeren Rechenausdrücken ein. (Je geübter man darin ist, umso stärker zeigt sich das.)

Wer noch unerfahren in der Berechnung von NPN- und UPN-Ausdrücken ist, sollte sich Zwischenstände zu begonnenen (und noch nicht abgeschlossenen) Operationen und Zwischenergebnissen auf ein Notizblatt schreiben, bis das Endergebnis erreicht ist. In der Regel sind die beiden Schreibweisen so recht schnell durchschaut und erlernt. Doch braucht es ein bißchen Übung (und anfänglich ein Notizblatt) dazu.

1.3. Termbäume verstehen

Willst Du einen (Dir eventuell unverständlichen) mathematischen Ausdruck (Term) berechnen, ist es gut, zuerst dessen Struktur zu bestimmen. Damit erleichterst Du Dir dessen Berechnung. Und auch, wenn Du gerne einen Taschenrechner nutzt, solltest Du zunächst die Struktur des Terms verstehen. Denn beim Eintippen in einem Taschenrechner sind manchmal zusätzliche Rechenklammern einzubringen oder andere Dinge zu beachten.

Ohne ein Verständnis der Struktur passieren beim Eintippen rasch Fehler, die man nicht erkennt und möglicherweise auch nicht versteht. Das Verstehen mathematischer Strukturen hilft Dir also beim Selberrechnen, als auch beim Einsatz von Rechenhilfen wie Computer oder Taschenrechner.

Die Strukturen von Termen sind leicht zu erkennen, wenn Du in Rechenbäumen denkst oder Dir diese behelfsweise skizzierst. Dabei gilt, daß jeder Term - egal wie kompliziert er ist - als Rechenbaum skizziert werden kann, und als Rechenbaum ganz leicht zu lösen ist.

Die Struktur eines Terms kannst Du immer an seinem Rechenbaum ablesen. Die diesbezüglichen Fertigkeiten kannst Du an einfachen Beispielen üben. Die nachstehenden Abschnitte formulieren dazu kleine Abschnitte und Beispiele, die erklären, wie Du diese mittels Rechenbäume (Termbäume) strukturieren und lösen kannst.

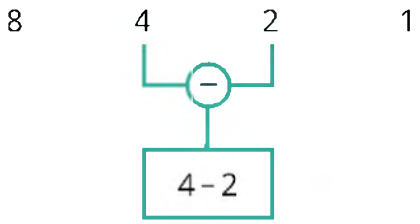
1.3.1. Beispiel 1: Term ohne Variable

Bestimme die Struktur des Terms $8:(4-2)-1$ **.**

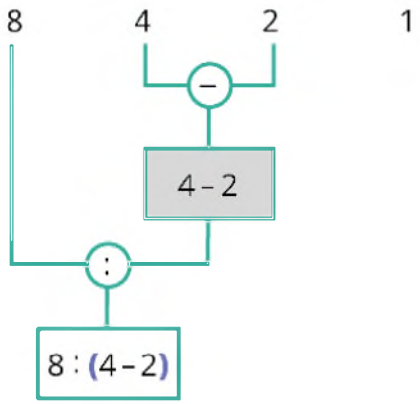
So gehst Du vor: ***Stelle den Rechenbaum auf. Bestimme dazu die Reihenfolge, in der die Rechenoperationen ausgeführt werden müssen. Nutze dabei die Vorrangregeln.***

Beachte, daß die Operatoren ihre Operanden stets in der richtigen Reihenfolge (von links nach rechts) erhalten, und die Reihenfolge beim Skizzieren nicht vertauscht wird.

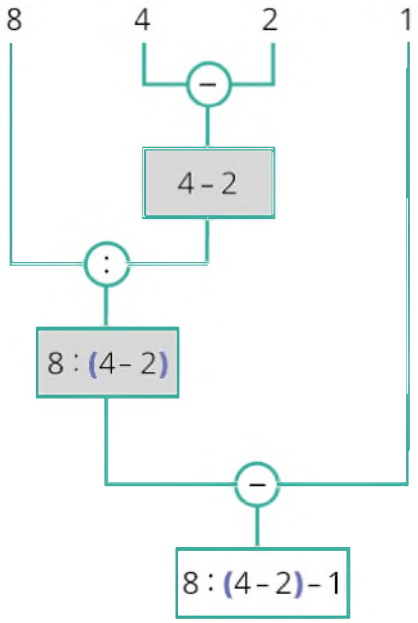
Als **erstes**: Klammern $(4-2)$.



Als **zweites**: Division $8:(4-2)$.



Als **drittes**: Subtraktion $\mathbf{8:(4-2)-1}$.



Vorrangregeln sind:

- Klammern immer zuerst
- Potenzen ausrechnen
- Punkt- vor Strichrechnung
- Von links nach rechts rechnen

In Wortform lautet der Term:

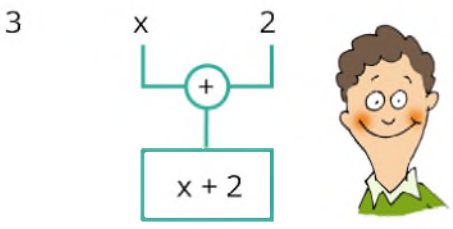
Dividiere 8 durch die Differenz von 4 und 2 und subtrahiere 1 vom Quotienten.

1.3.2. Beispiel 2: Term mit Variable

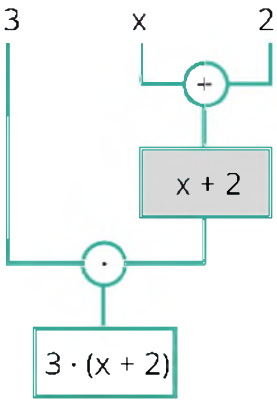
Bestimme die Struktur des Terms $\mathbf{3\cdot(x+2)}$ **.**

So gehst Du vor: ***Stelle den Rechenbaum auf. Bestimme dazu die Reihenfolge, in der die Rechenoperationen ausgeführt werden müssen. Nutze dabei die Vorrangregeln.***

Als **erstes**: Klammern $\mathbf{(x+2)}$.



Als **zweites**: Multiplikation $3 \cdot (x+2)$.



Vorrangregeln sind:

- Klammern immer zuerst
- Potenzen ausrechnen
- Punkt- vor Strichrechnung
- Von links nach rechts rechnen

In Wortform lautet der Term:

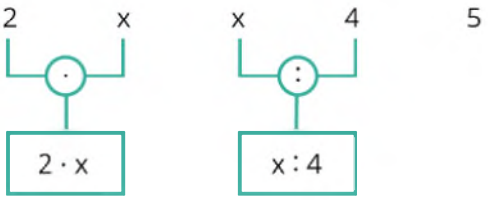
Multipliziere 3 mit der Summe von x und 2.

1.3.3. Beispiel 3: Term mit Variable

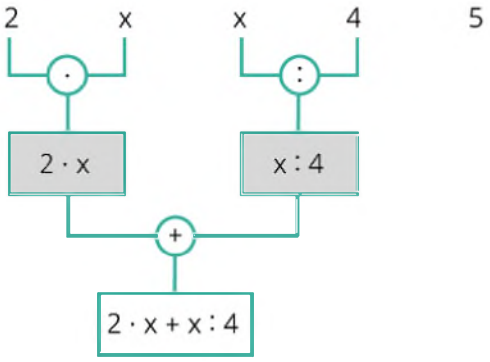
Bestimme die Struktur des Terms $2 \cdot x + x : 4 + 5$.

So gehst Du vor: ***Stelle den Rechenbaum auf. Bestimme dazu die Reihenfolge, in der die Rechenoperationen ausgeführt werden müssen. Nutze dabei die Vorrangregeln.***

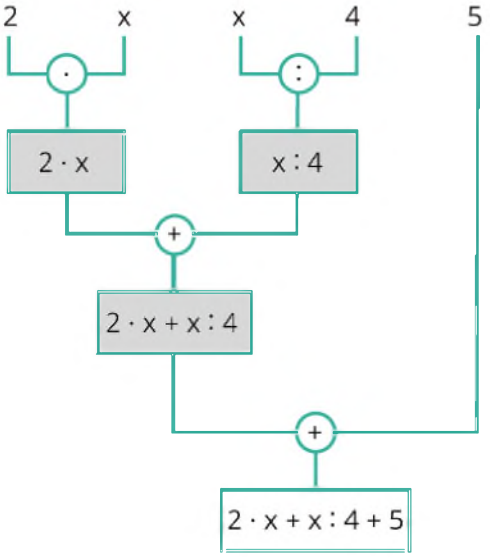
Als **erstes**: Multiplikation und Division $2 \cdot x$ und $x : 4$.



Als **zweites**: Addition $2 \cdot x + x : 4$.



Als **drittes**: Addition $2 \cdot x + x : 4 + 5$.



Vorrangregeln sind:

- Klammern immer zuerst
- Potenzen ausrechnen
- Punkt- vor Strichrechnung
- Von links nach rechts rechnen

In Wortform lautet der Term:

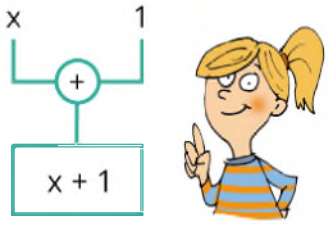
Addiere das Produkt von 2 und x zum Quotienten aus x und 4. Addiere anschließend 5.

1.3.4. Beispiel 4: Term mit Potenzen

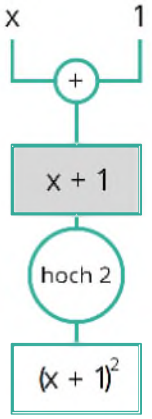
Bestimme die Struktur des Terms.

So gehst Du vor: ***Stelle den Rechenbaum auf. Bestimme dazu die Reihenfolge, in der die Rechenoperationen ausgeführt werden müssen. Nutze dabei die Vorrangregeln.***

Als **erstes**: Klammern $(x+1)$.



Als **zweites**: Potenzen $(x+1)^2$.



Vorrangregeln sind:

- Klammern immer zuerst
- Potenzen ausrechnen
- Punkt- vor Strichrechnung
- Von links nach rechts rechnen

In Wortform lautet der Term:

Bilde das Quadrat der Summe von x und 1.

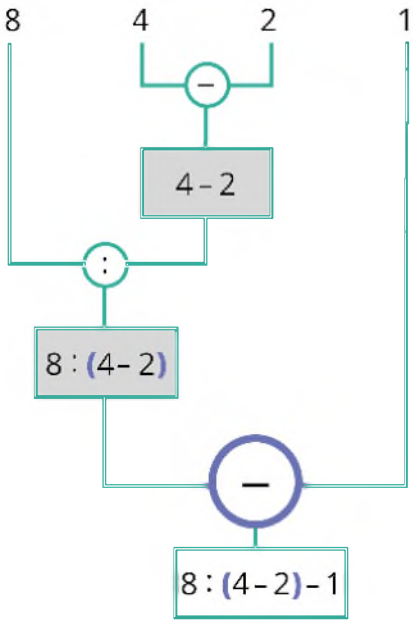
1.3.5. Den Typ eines Terms erkennen

Den **Typ** eines Terms erkennst Du an der **letzten Rechenoperation** im Rechenbaum.

Letzte Rechenoperation	Typ des Terms
Addition	Summe
Subtraktion	Differenz
Multiplikation	Produkt
Division	Quotient
Potenzierung	Potenz

1.3.5.1 Typ erkennen - Beispiel 1: Term ohne Variable

Hier ist nochmal der Rechenbaum des Terms $\mathbf{8:(4-2)-1}$. Die letzte Rechenoperation steht in der untersten Verzweigung.

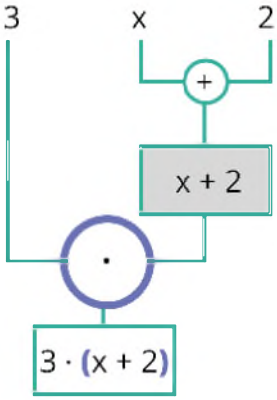


Die letzte Rechenoperation ist eine **Subtraktion**. Also ist der Typ des Terms eine **Differenz**.

Du sagst: »Der Term ist eine Differenz«.

1.3.5.2 Typ erkennen - Beispiel 2: Term mit Variable

Hier ist nochmal der Rechenbaum des Terms $3 \cdot (x+2)$. Die letzte Rechenoperation steht in der untersten Verzweigung.



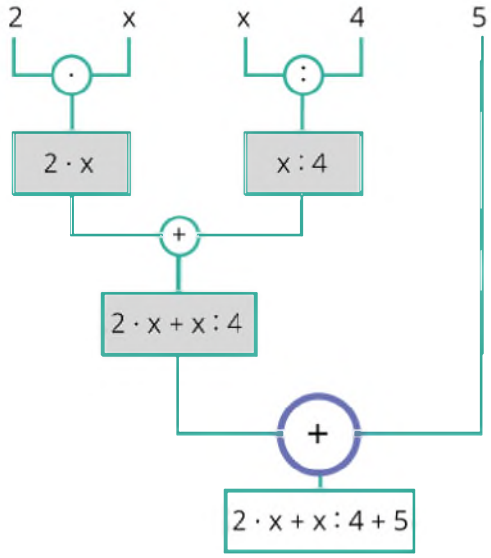
Die letzte Rechenoperation ist eine Multiplikation. Also ist der Typ des Terms eine **Produkt**.

Du sagst: »Der Term ist ein Produkt«.

Der ganz zu unterst liegende Element eines Rechenbaums wird **Wurzel** (des Rechenbaums) genannt und liefert das Endergebnis des Terms.

1.3.5.3. Typ erkennen - Beispiel 3: Term mit Variable

Hier ist nochmal der Rechenbaum des Terms $2 \cdot x + x : 4 + 5$. Die letzte Rechenoperation steht in der untersten Verzweigung.

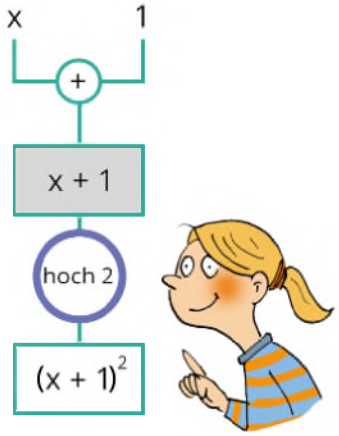


Die letzte Rechenoperation ist eine **Addition**. Also ist der Typ des Terms eine **Summe**.

Du sagst: »Der Term ist eine Summe«.

1.3.5.4. Typ erkennen - Beispiel 4: Term mit Potenz

Hier ist nochmal der Rechenbaum des Terms $(x+1)^2$. Die letzte Rechenoperation steht in der untersten Verzweigung.

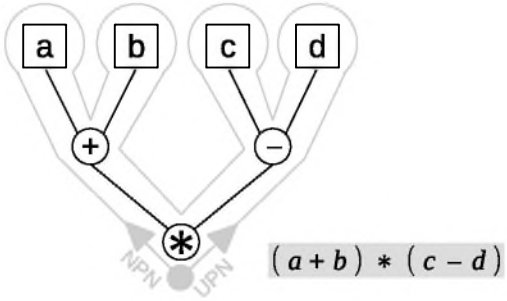


Die letzte Rechenoperation ist eine **Potenzierung**. Also ist der Typ des Terms eine **Potenz**.

Du sagst: »Der Term ist eine Potenz«.

1.4. Termbäume (Rechenbäume)

Rechenausdrücke werden in der Mathematik ***Terme*** genannt, und jeder Term (Rechenausdruck) kann in *Infix-Notation* (Schulmathematik), in *Präfix-Notation* (*NPN*), in *Postfix-Notation* (*UPN*) oder als *Termbaum* (Rechenbaum) niedergeschrieben werden.



UPN: $a\ b + c\ d - *$
NPN: $* + a\ b - c\ d$

Die Art der gewählten Notation ändert nicht den Term, sondern legt lediglich die Schreibweise fest, in der ein Term in einer Schriftsprache oder in einer Strukturdarstellung niedergeschrieben wird. Jede der vier genannten Schreibweisen ist eine reguläre mathematische Schreibweise, und ihre Kenntnis erweitert die eigenen mathematischen Ausdrucksmöglichkeiten.

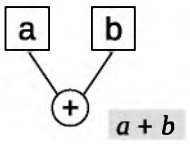
Ganz allgemein solltest Du Dich für diejenige Schreibweise entscheiden, die Dir für den vorliegenden Zweck die meisten Vorteile bietet. So bedeutet ein Termbaum zwar einen höheren Konstruktionsaufwand, bietet Dir dafür aber die höchstmögliche Anschaulichkeit, wie ein Term abzuarbeiten ist; zudem kannst Du die Präfix-Notation (NPN) als auch die Infix-Notation (UPN) ganz einfach aus einem Termbaum ermitteln. Dazu gehst Du wie folgt vor:

- Für die ***Präfix-Notation*** (***NPN***) beginnst Du an der Wurzel eines Termbaumes und fährst dessen Umriß im Uhrzeigersinn ab. Du notierst jeden Operanden und jeden Operator, auf den Du triffst, in einer ("endlosen") Textzeile von links nach rechts (also von vorne nach hinten); dabei darf jeder Operand und jeder Operator nur einmal in der notierten Textzeile niedergeschrieben werden.
- Für die ***Postfix-Notation*** (***UPN***) beginnst Du an der Wurzel eines Termbaumes und fährst dessen Umriß entgegen dem Uhrzeigersinn ab. Du notierst jeden Operanden und jeden Operator, auf den Du triffst, in einer ("endlosen") Textzeile von rechts nach links (also von hinten nach vorne); dabei darf auch hier jeder Operand und jeder Operator nur einmal in der notierten Textzeile niedergeschrieben werden.

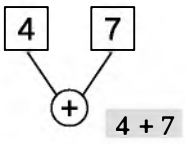
Übung:

Decke mit einem Blatt Papier die nachstehenden UPN- und NPN-Schreibweisen komplett ab und führe dann für die Termbäume einer Zeile den UPN- und den NPN-Termbaumumlauf durch, und notiere dabei die zu jedem Termbaum die zugehörige Postfix-Notation (UPN) und Präfix-Notation (NPN).

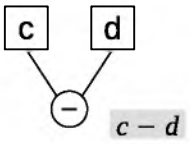
Vergleiche anschließend direkt, ob Deine Umwandlungsergebnisse zu den jeweiligen Termbäumen einer Zeile richtig sind, oder wo Deine Denkfehler lagen. Das abdeckende Papierblatt schiebst Du dazu tiefer, so daß Dir die korrekten Ergebnisse für in der jeweiligen Termbäume-Zeile angezeigt wird.



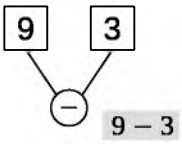
UPN: $a\ b\ +$
NPN: $+\ a\ b$



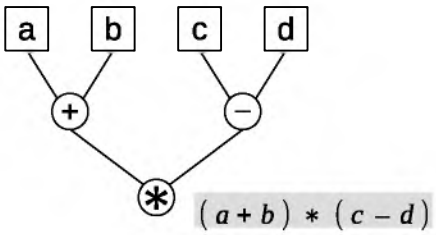
UPN: $4\ 7\ +$
NPN: $+\ 4\ 7$



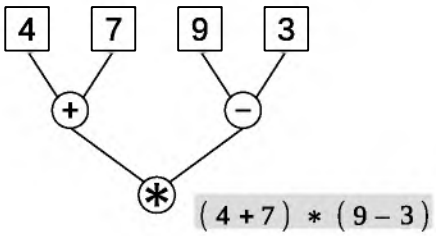
UPN: $c\ d\ -$
NPN: $-\ c\ d$



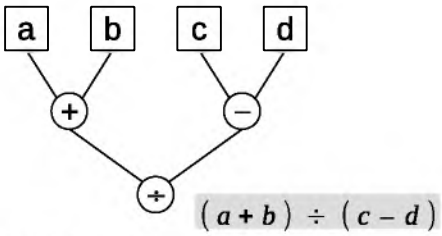
UPN: $9\ 3\ -$
NPN: $-\ 9\ 3$



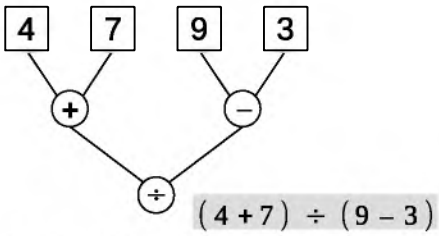
UPN: $a\ b\ +\ c\ d\ -\ *$
NPN: $*\ +\ a\ b\ -\ c\ d$



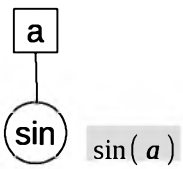
UPN: $4\ 7\ +\ 9\ 3\ -\ *$
NPN: $*\ +\ 4\ 7\ -\ 9\ 3$



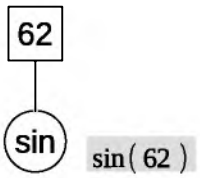
UPN: $a\ b\ +\ c\ d\ -\ \div$
NPN: $\div\ +\ a\ b\ -\ c\ d$



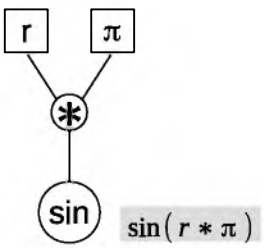
UPN: $4\ 7\ +\ 9\ 3\ -\ \div$
NPN: $\div\ +\ 4\ 7\ -\ 9\ 3$



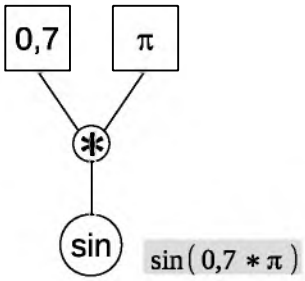
UPN: a sin
NPN: sin a



UPN: 62 sin
NPN: sin 62



UPN: $r\ \pi\ *$ sin
NPN: sin $*\ r\ \pi$



UPN: $0{,}7\ \pi\ *$ sin
NPN: sin $*\ 0{,}7\ \pi$

1.4.1. Termbäume erstellen

Auch wenn wir bisher nur Termbäume betrachtet haben, die von oben nach unten skizziert und gelöst werden, so ist letztlich nicht festgelegt, ob man sie

- *von oben nach unten hin* skizziert und löst, oder
- *von unten nach oben hin*.

Jedenfalls sind beide Varianten zulässig und äquivalent. Wir zeigen dies am Beispiel der Zinsesszinsrechnung. Die Formel zur Zinszinsberechnung lautet:

$$K_n = K_0 \cdot \left(1 + \frac{p}{100}\right)^n$$

Dabei ist

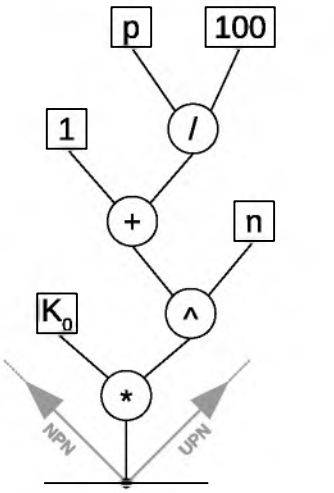
K_n - das Endkapital nach n Jahren Laufzeit

K_0 - das Anfangskapital zu Beginn der Laufzeit

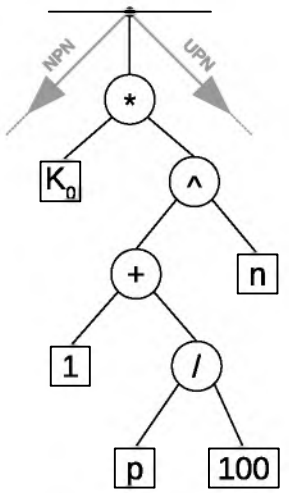
p - der Zinssatz in Prozent (z.B. *3,7*)

n - die Laufzeit in Anzahl an Jahren

Als nächstes werden wir die beiden Turmbaumvarianten notieren, die sich aus der Formel zur Zinsesszinsberechnung ergeben. Für die Potenzrechnung werden wir dabei das Symbolzeichen ^ nutzen:



Termbaum ist
von oben nach unten
abzuarbeiten



Termbaum ist
von unten nach oben
abzuarbeiten

Zu beiden Termbäumen jeweils der Startpunkt des Termbaum-Umlaufs für die zugehörige UPN- bzw. NPN-Schreibweise angegeben. Die gemeinsame Merkregel zu diesen Termbäumen (unabhängig davon, ob sie von oben-nach-unten-hin oder von unten-nach-oben-hin angelegt sind) lautet:

- Der **NPN-Umlauf** beginnt an der Wurzel des Termbaumes und fährt *zunächst links* um diesen herum, und erreicht dann über dessen rechte Seite wieder die Wurzel. Die **NPN-Schreibweise** wird dabei in einer "endlosen" Zeile von *links nach rechts* (d.h. von vorne nach hinten) notiert.

  $$* \quad K_0 \quad \wedge \quad + \quad 1 \quad / \quad p \quad 100 \quad n$$

- Der **UPN-Umlauf** beginnt an der Wurzel des Termbaumes und fährt *zunächst rechts* um diesen herum, und erreicht dann über dessen linke Seite wieder die Wurzel. Die **UPN-Schreibweise** wird dabei in einer "endlosen" Zeile von *rechts nach links* (d.h. von hinten nach vorne) notiert.

  $$K_0 \quad 1 \quad p \quad 100 \quad / \quad + \quad n \quad \wedge \quad *$$

1.5. Die besondere Bedeutung der Postfix-Notation (UPN)

Postfix-Notation (***UPN***) ist häufig die Grundlage für die interne Arbeitsweise von Taschenrechnern, Computern, Computerprogrammen und Programmiersprachen, selbst wenn diese von einem Menschen, der sie benutzt, Infix-Eingaben entgegennehmen und zurückgeben. Neben den allgemein bekannten Infix-Taschenrechnern gibt es Taschenrechner am Markt, die Eingaben in Postfix (UPN) entgegennehmen (etwa Taschenrechner von *HP* oder auch *SwissMicros*).

Programmiersprachen, die (zumindest intern) UPN nutzen sind:

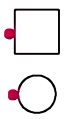
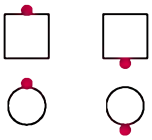
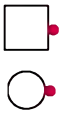
- *Forth*,
- *Postscript*,
- *Factor*,
- *C* / *C++* (jeweils vor der Registeroptimierung),
- *Reverse Polish LISP* *(RPL)*,
- und andere ...

Der Grund dafür ist, daß Postfix-Notationen mit einfacher Technik und minimalem Aufwand realisiert werden können und somit auch kostengünstig sind; zudem sind sie eindeutig und Fehler (wie auch die Fehlerstellen) können direkt erkannt werden. Das macht sie für Entwickler von Hardware (Bauteile und Geräte), Programmiersprachen und Softwaresystemen interessant.

1.6. Verbessertes Umlaufverfahren

Das bislang genutzte Termumlauf läßt sich verbessern, wenn man die Objekte des Termbaums (Operanden und Operatoren) mit einer einheitlichen Markierung versehen werden. Je nach Position dieser Markierung dann ein Termbaum damit in die *Präfix-* , *Infix-* oder *Postfix-Notation* umgewandelt werden.

Als Markierung nutzen wir einen Punkt (•) und bringen diesen wie folgt an:

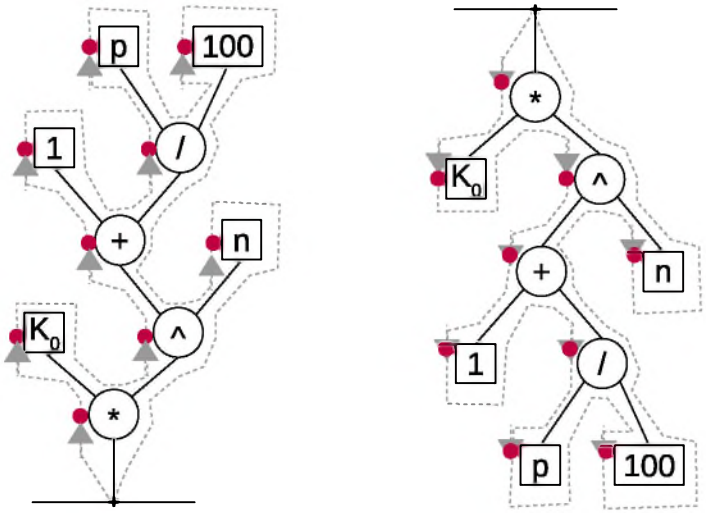
Für eine Umwandlung in die ...		
Präfix-Notation (NPN)	**Infix-Notation**	**Postfix-Notation** (UPN)
		
Operanden und Operatoren werden **links** markiert (d.h. "davor").	Operanden und Operatoren werden **oben** / **unten** markiert (d.h. "mittig").	Operanden und Operatoren werden **rechts** markiert (d.h. "dahinter").

Die Markierung erlaubt jetzt, daß diese drei Notationen nach einer gemeinsamen Umlauf-Regel erstellt werden können:

- Hier beginnt **jeder Umlauf** an der Wurzel des Termbaums und fährt *zunächst links* um diesen herum, und erreicht dann über dessen rechte Seite wieder die Wurzel. Dabei werden die gesetzten **Markierungspunkte angefahren** und der Reihe nach in einer "endlosen" Zeile von *links nach rechts* (d.h. von vorne nach hinten) notiert. Durch die einheitliche Lage der Markierungspunkte (links, rechts oder mittig) ergibt sich dabei die gewünschte Notation.

1.6.1. Verbesserter NPN-Umlauf

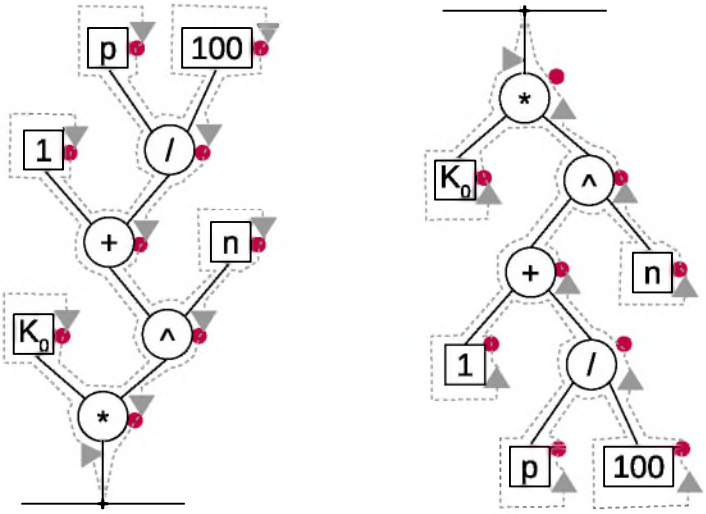
Hier jeweils ein Beispiel für den verbesserten NPN-Umlauf bei "stehenden" und beim "hängenden" Termbaum der Zinsesszinsberechnung:



$$* \quad K_0 \quad \wedge \quad + \quad 1 \quad / \quad p \quad 100 \quad n$$

1.6.2. Verbesserter UPN-Umlauf

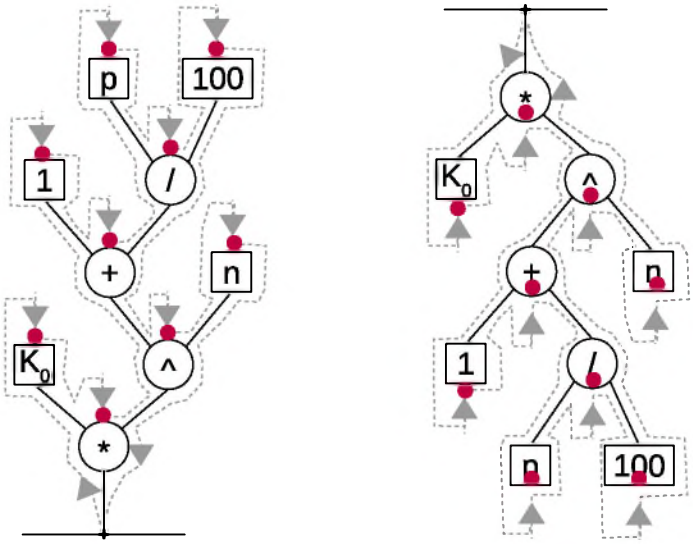
Hier jeweils ein Beispiel für den verbesserten UPN-Umlauf bei "stehenden" und beim "hängenden" Termbaum der Zinsesszinsberechnung:



$$K_0 \quad 1 \quad p \quad 100 \quad / \quad + \quad n \quad \wedge \quad *$$

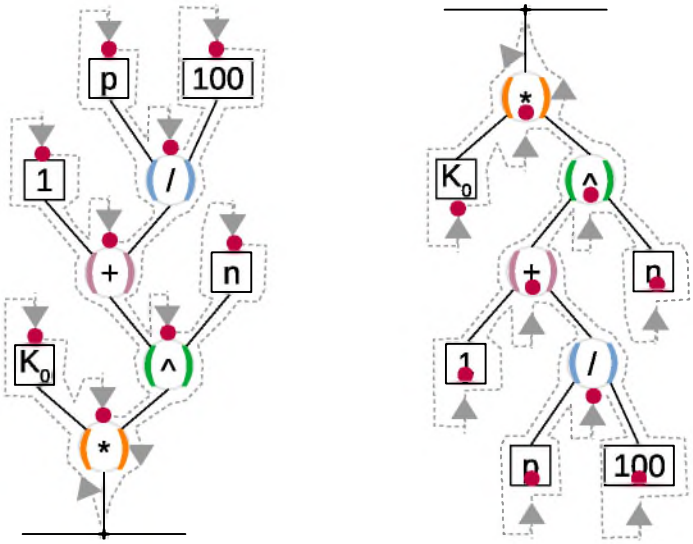
1.6.3. Die Verbesserung bietet Infix-Umlauf

Sämtliche Markierungspunkte werden so platziert, daß sie sich auf der zur Wurzel des Termbaums entgegengesetzten Seite befinden.



Im Unterschied zur Präfix-Notation (NPN) und zur Postfix-Notation (UPN) müssen bei der Infix-Notation Vorrangigkeiten und Klammersetzungen beachtet werden. Diese lassen sich ganz einfach erstellen, wenn **zusätzlich** (zu den Markierungspunkten) **jedem Operator ein Klammerpaar zuordnen**.

Aus Lehrgründen und zur besseren Veranschaulichung sind diese Klammerpaare im nachfolgenden Abschnitt jeweils mit einer eigen Farbe versehen:



Führt uns der Umlauf nun an eine öffnende Klammer "(" vorbei, tragen wir diese in die "endlose" Infix-Zeile ein; führt uns der Umlauf nun an eine schließende Klammer ")" vorbei, so tragen wir diese in die "endlose" Infix-Zeile ein; und treffen wir auf einen Markierungspunkt, so tragen wir den zugehörigen Operator oder Operanden in die "endlose" Infix-Zeile ein. Die "endlose" Infix-Zeile wird dabei von *links nach rechts* (d.h. von vorne nach hinten) notiert. Wir erhalten:

$$(K_0 * ((1 + (p / 100)) ^ n))$$

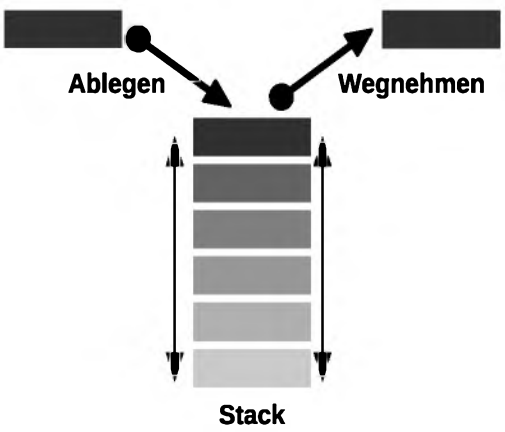
Anschließend hilft uns ein bißchen Nachdenken dabei jene Klammerpaare zu finden, die wir bedenkenlos entfernen können, ohne die mathematische Korrektheit des Infix-Term zu stören. Wir erhalten:

$$K_0 * (1 + p / 100) ^ n$$

Mit hinreichend Übung kann das Eintragen der Klammerpaare entfallen. weil man dann um diesen Vorgang weiß und ihn rein gedanklich durchführt, wo immer man im Termbaum auf einen Operator trifft.

1.7. Der Stack (Stapelspeicher)

Ein ***Stack*** (auch ***Stapelspeicher*** oder ***Kellerspeicher*** genannt) ist eine einfache Datenstruktur, die gerne zur Abarbeitung von Postfix-Termen genutzt wird.



Ein Stack beruht auf einem Speicherkonzept, mit genau zwei Operationen:

- Ein Element auf dem Stack (Stapel) ablegen (*push* genannt),
- ein Element von dem Stack (Stack) wegnehmen (*pop* genannt);

wobei immer nur eine Operation aufgeführt werden darf. Sind mehrere Operationen auszuführen, so werden diese nacheinander durchgeführt, jedoch niemals zeitgleich.

Die auf den Stack (Stapel) abgelegten Elemente werden dabei konzeptionell aufeinandergestapelt. Das zuerst auf dem Stack abgelegte Element liegt ganz unten; das als zweites abgelegte Element liegt auf den ersten; das als drittes abgelegte Element liegt auf dem zweiten, usw. Und weggenommen werden kann jeweils nur das zu oberst liegende Element, als dasjenige, das zum Zeitpunkt des Wegnehmens das letzte auf dem Stack abgelegte Element ist. Das jeweils zu oberst liegende Stack-Element wird "***Top of Stack***" (***TOS***) genannt.

Manche Programmiersprache enthält gar einen Befehl, der prüft, ob deren Stack leer ist oder nicht (also daß kein Element auf dem Stack abgelegt ist). So ist der Stack leer, wenn die Pop-Operation zurückmeldet, daß sie kein Element vom Stack abheben konnte.

Auch der *Shunting-Yard-Algorithmus*, den wir im nächsten Abschnitt kennenlernen werden, nutzt einen Stack.

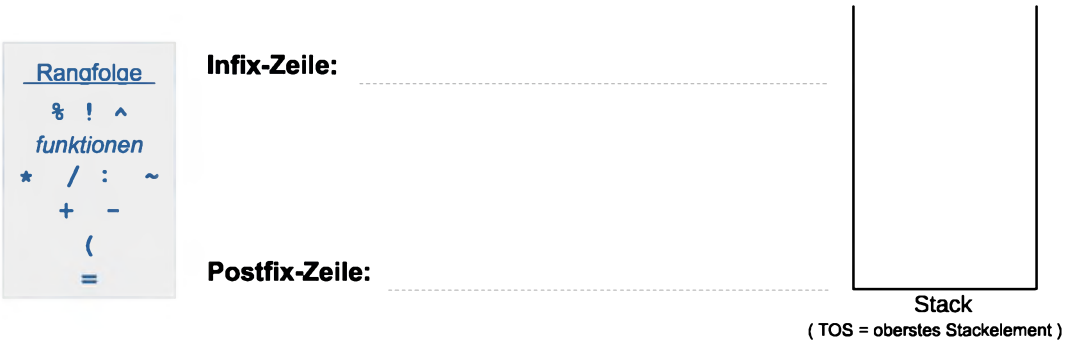
1.8. Der Shunting-Yard-Algorithmus

Von INFIX zu POSTFIX umformen

Der ***Shunting-Yard-Algorithmus*** (auch ***"Rangierbahnhof"-Algorithmus*** genannt) ist eine Methode, die mathematische Terme von der Infix-Notation in die Postfix-Notation überführt, ohne daß als Zwischenschritt ein Termbaum zu notieren und zu umfahren ist. Sein Name rührt daher, daß der Ablauf des Algorithmus an einen Rangierbahnhof erinnert. Gefunden und veröffentlicht wurde der Algorithmus wurde von dem bedeutenden niederländischen Informatiker ***Edsger W. Dijkstra*** (*1930, †2002).

Der Algorithmus nutzt:

- eine Rangliste (Rangfolge),
- einen Stack (Stapelspeicher), und
- zwei "endlose" Textzeilen (*Queue*) für Infix und Postfix.



mit:

- **%** für Prozentangaben,
- **^** für Potenz,
- **(** für Klammersetzung.
- **!** für Fakultät,
- **~** für Modulo-Division (Division mit Rest),

Anders als bei der mathematischen Rangfolge werden Klammersetzungen von dem Algorithmus gesondert behandelt, weswegen die Klammersetzung hier auf die vorletzte Position (über dem Gleichheitszeichen) wandert.

Jeder Infix-Term wird zuvor zu einen einzeiligen Infix-Term umgeformt, so daß dieser etwa in die Eingabezeile eines Computerprogramms eingegeben werden könnte. Für den Algorithmus wird dieser einzeilige Infix-Term in die "endlose" Infix-Zeile eingetragen.

Diese "endlose" Infix-Zeile wird anschließend von links nach rechts Element für Element abgearbeitet und zwar mit jedem Schleifendurchlauf des Algorithmus.

- Nimm ein Element. Wenn es ...
  - eine Zahl (Konstante, Variable) ist, dann trage sie in die Liste ein.
  - ein Operator (Funktion) ist, ...
    - prüfe, ob TOS ein höherrangiger Operator ist, wenn ja verschiebe diesen in die Liste.
    - Nun lege den aktuellen Operator auf den Stack.
  - eine öffende Klammer "(" ist, lege sie auf den Stack ab.
  - eine schließende Klammer ")" ist, ...
    - prüfe, ob TOS keine öffnende Klammer "(" ist, ist das der Fall verschiebe TOS in die Liste.
    - Wiederhole das solange, bis der Stack leer ist, oder TOS eine öffnende Klammer "(" ist; dann nimm diese vom Stack und verwerfe sie.
- Wiederhole den ganzen Durchlauf bis alle Elemente verarbeitet sind. Was am Ende noch auf dem Stack liegen sollte, verschiebe nacheinander in die Liste (pop).

Dazu ein Beispiel:

Wir greifen auf die Infix-Formel zur Zinsesszinsrechnung zurück.Sie lautet:

$$K_n = K_0 \cdot \left(1 + \frac{p}{100}\right)^n$$

Dabei ist

K_n - das Endkapital nach n Jahren Laufzeit

K_0 - das Anfangskapital zu Beginn der Laufzeit

p - der Zinssatz in Prozent (z.B. *3,7*)

n - die Laufzeit in Anzahl an Jahren

Da uns für die Berechnung ausschließlich die rechte Seite der Formel interessiert, formen wir diese zu einer einzeiligen Infix-Term (Infix-Ausdruck) um. Dieser lautet:

$$K_0 \ * \ (\ 1 \ + \ p \ / \ 100 \ ) \ \wedge \ n$$

Diesen Infix-Term tragen wir nun in die "endlose" Infix-Zeile des *Shunting-Yard-Algorithmus* ein. Damit sind dessen notwendige Vorarbeiten abgeschlossen, und wir können mit dem ersten Durchlauf des Algorithmus beginnen:

1) **Infix-Zeile:** $K_0 \ * \ (\ 1 \ + \ p \ / \ 100 \ ) \ \wedge \ n$

Erstes Element, das wir der Infix-Zeile entnehmen werden.

Das erste Element ist eine Variable und wird daher in die Postfix-Zeile eingetragen.

Postfix-Zeile: K_0

Stack

2) **Infix-Zeile:** $* \ (\ 1 \ + \ p \ / \ 100 \ ) \ \wedge \ n$

Das nächste Element ist ein Operator und wird auf den Stack verschoben (abgelegt).

Postfix-Zeile: K_0

Stack: *

3) **Infix-Zeile:** $(\ 1 \ + \ p \ / \ 100 \ ) \ \wedge \ n$

Die öffnende Klammer kommt auf den Stack.

Postfix-Zeile: K_0

Stack: (*

4) **Infix-Zeile:** $1 \ + \ p \ / \ 100 \ ) \ \wedge \ n$

Die Zahl wird in die Postfix-Zeile eingetragen.

Postfix-Zeile: $K_0 \ 1$

Stack: (*

5) **Infix-Zeile:** + p / 100) ^ n

Der Plus-Operator kommt auf den Stack.

Postfix-Zeile: K_0 1

6) **Infix-Zeile:** p / 100) ^ n

Die Variable wird in die Postfix-Zeile eingetragen.

Postfix-Zeile: K_0 1 p

7) **Infix-Zeile:** / 100) ^ n

Der Operator kommt auf den Stack.

Postfix-Zeile: K_0 1 p

8) **Infix-Zeile:** 100) ^ n

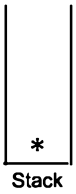
Die Zahl wird in die Postfix-Zeile eingetragen.

Postfix-Zeile: K_0 1 p 100

9) **Infix-Zeile:**) ^ n

Die schließenden Klammer räumt den Stack teilweise ab.

Postfix-Zeile: K_0 1 p 100 / +



10) **Infix-Zeile:** ^ n

Der Operator kommt auf den Stack.

Postfix-Zeile: K_0 1 p 100 / +

11) **Infix-Zeile:** n

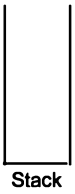
Die Variable wird in der Postfix-Zeile eingetragen.

Postfix-Zeile: K_0 1 p 100 / + n

12) **Infix-Zeile:**

Am Ende ist die Infix-Zeile leer, und der verbliebene Stack wird abgeräumt.

Postfix-Zeile: K_0 1 p 100 / + n ^ *



Wir erhalten schließlich den Postfix-Term (Postfix-Ausdruck):

$$K_0 \quad 1 \quad p \quad 100 \quad / \quad + \quad n \quad ^ \quad *$$

Mittels des Shunting-Yard-Algorithmus haben wir nun eine korrekte UPN-Notation zu dem obigen Infix-Ausdruck erhalten.

Hinweis: Mitunter ergeben sich aus einem korrekt durchgeführten Termbaum-Umlauf und aus einem korrekt durchgeführten Shunting-Yard-Algorithmus unterschiedliche UPN-Notationen für den gleichen Infix-Ausdruck. Darin liegt jedoch kein Widerspruch, denn diese UPN-Notationen sind zueinander äquivalent – und sind somit allesamt korrekt.

1.9. Weitere Umformungen

1.9.1. von POSTFIX zu PRÄFIX

1.9.2. von POSTFIX zu INFIX

Beide Umwandlungen nutzen einen Stack. Die vorliegende Postfix-Notation wird eine eine "endlose" Postfix-Zeile eingetragen, und auch die Ergebnis-Notation wird in einer "endlosen" Ergebnis-Zeile eingetragen. Bei beiden Umwandlungen wird die "endlose" Postfix-Zeile von links nach rechts Element um Element abgearbeitet.

Die Algorithmen

- Lesen des Postfix-Ausdrucks von links nach rechts.
- Wenn das Element ein Operand (Zahl, Variable) ist, lege es auf den Stack ab.
- Wenn das Element ein Operator oder eine Funktion ist, werden die Anzahl an Operanden (Zahlen, Variablen) vom Stack genommen, die der Operator (oder die Funktion) benötigt.

  Erzeuge einen Präfix-Ausdruck, indem die Operanden und den Operator/Funktion miteinander verknüpft sind:

  z.B.: Operator *Operand2* *Operand1*
  + *x* 7

  Und lege diesen Präfix-Ausdruck zurück auf dem Stack. Dieser Präfix-Ausdruck ist danach wie eine einzelne Zahl zu behandeln.
- Wiederholen Sie die obigen Schritte bis zum Ende des Postfix-Ausdrucks.
- Nimm nun das verbliebene Element vom Stack und trage es in die "endlose" Ergebnis-Zeile ein.

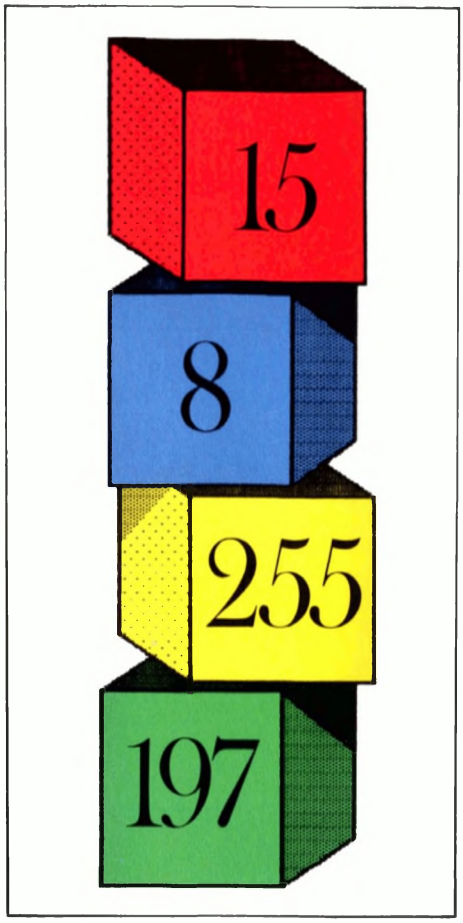
- Lesen des Postfix-Ausdrucks von links nach rechts.
- Wenn das Element ein Operand (Zahl, Variable) ist, lege es auf den Stack ab.
- Wenn das Element ein Operator oder eine Funktion ist, wird die Anzahl an Operanden (Zahlen, Variablen) vom Stack genommen, die der Operator (oder die Funktion) benötigt.

  Erzeuge einen Infix-Ausdruck, indem die Operanden und den Operator/Funktion miteinander verknüpft sind:

  z.B.: *Operand2* Operator *Operand1*
  x **+** 7

  Und lege diesen Infix-Ausdruck zurück auf dem Stack. Dieser Infix-Ausdruck ist danach wie eine einzelne Zahl zu behandeln.
- Wiederholen Sie die obigen Schritte bis zum Ende des Postfix-Ausdrucks.
- Nimm nun das verbliebene Element vom Stack und trage es in die "endlose" Ergebnis-Zeile ein.

1.10. Der Stack



„Stacks" oder Stapelspeicher spielen in der Datenverarbeitung für das kurzfristige Deponieren von Informationen eine wichtige Rolle. Sie sind besonders einfach zu handhaben. Beim LIFO-Stack ist wie beim Würfelstapel aus dem Baukasten der Zugriff nur von oben her möglich. Die Reihenfolge der gespeicherten Größen bleibt vom Eintragen bis zum Auslesen unverändert, was für die Verwaltung der Rücksprungadressen bei verschachtelten Unterprogrammaufrufen ideal ist. Eine einfache Einsatzmethode für nicht so routinierte Anwender.

Stack = Stapelspeicher

Der „Stack" ist als spezieller Speicherbereich der CPU eines Computers zugeordnet und wird zur kurzfristigen Ablage von Registerinhalten und anderen Informationen benutzt. Die Kapazität des Stack und seine Lage im Arbeitsspeicher hängt vom jeweiligen Prozessor ab: Beim 6502 liegt der Stack auf der Nullseite (Zero Page) und faßt nur 256 Byte, beim Z80 dagegen können ihm beliebige Adressen und bis zu 64 KByte zugeteilt sein. Der Stack ist aber stets in gleicher Weise organisiert und geordnet.

Jeder Prozessor verfügt über besondere Stack-Befehle zum Eintragen (PUSH) und Entnehmen (POP) von Information. Beim üblichen LIFO-Stack werden die Daten „aufeinander" gestapelt; die letzten Einträge werden zuerst ausgelesen.

An diesem Prinzip kommt man beim Umgang mit dem Stack kaum vorbei: Die tieferstehende Information ist nur zugänglich, wenn der Stack von oben her weit genug abgearbeitet ist. Für den Zugriff hält ein bestimmtes CPU-Register, der „Stack Pointer" (Stapelzeiger), stets als aktuelle Adresse die der zuletzt angesprochenen Speicherzelle bereit. Der Stapelzeiger wandert beim Füllen und Leeren des Stack automatisch auf- bzw. abwärts, kann aber seitens des Programmierers bei Bedarf auch verschoben werden. Stack-Manipulationen sind nur im Maschinencode vorgesehen, aber es gibt sie auch in FORTH.

Quelle: *ComputerKurs*-Magazin, Heft 80, Seite 2241, Jahrgang 1984/1985

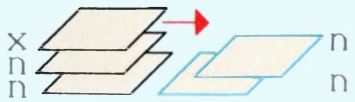
Rechnen mit UPN

2 + 3 gleich 2 3 +

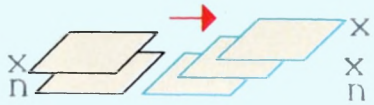
Gesteuerter Stapel

Es ist nicht immer möglich, den Stapel so aufzubauen, daß alle Zahlen in der richtigen Reihenfolge stehen. FORTH bietet daher eine Reihe von Wörtern, die die Werte des Stapels umstellen. Hier ein paar Beispiele:

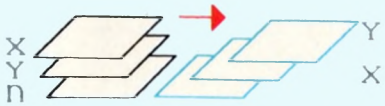
- **DROP** x–– (löscht den obersten Stapelwert)



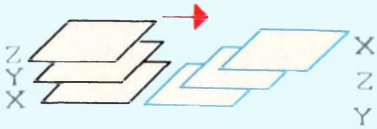
- **DUP** x––x, x (dupliziert den obersten Stapelwert)



- **SWAP** x, y––y, x (tauscht die beiden obersten Stapelelemente gegeneinander aus)

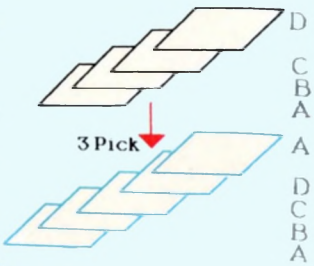


- **ROT** x, y, z –– y, z, x (rotiert die ersten drei Stapelelemente, so daß die zuunterst liegende Zahl oben liegt)

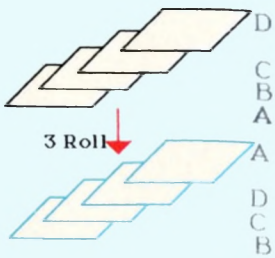


- **OVER** x, y –– x, y, x (kopiert die zweitunterste Nummer und legt sie oben auf den Stapel)

- **PICK** xn, . . . x2, x1, x0, n––xn, . . ., x2, x1, x0, xn (wie OVER. Sie müssen jedoch eine zusätzliche Zahl (n) angeben, die besagt, welche Zahl kopiert werden soll. 1 PICK entspricht beispielsweise OVER, 2 PICK kopiert die drittunterste Zahl, während 0 PICK wie DUP arbeitet)



- **ROLL** xn, . . ., x2, x1, x0, n –– . . ., x2, x1, x0, xn (wie ROT, aber mit einer zusätzlichen Zahl wie PICK. 2 ROLL entspricht ROT, während 3 ROLL vier Zahlen rotiert. 1 ROLL entspricht SWAP).



In FigFORTH fehlen PICK und ROLL. In FORTH-79 sind sie enthalten, doch spricht die Zusatzzahl einen um Eins höheren Wert an, als in FORTH-83. In FORTH-79 entspricht 3 ROLL daher ROT.

In dieser Folge untersuchen wir die Logik der „Umgekehrten Polnischen Notation" (UPN) und wollen zeigen, wie diese Schreibweise mathematische Abläufe und die Bearbeitung von Daten vereinfacht.

Es ist selbstverständlich, daß wir die Summe zweier Zahlen nur erhalten können, wenn wir beide Zahlen kennen. Bei der Eingabe von 2+3 (das "+" steht in der Mitte), wartet ein Computer daher erst auf die zweite Zahl, bevor er sich mit dem + beschäftigt. Die Maschine versteht die Aufgabe als 2 3 +.

Wir sind daran gewöhnt, mathematische Operatoren wie +, −, * und / in der Mitte („Infix-Notation") zu schreiben (zum Beispiel 2+2). Die Schreibweise entspricht der Umgangssprache (zwei plus zwei ergibt vier) und trennt die beiden Operanden deutlich voneinander.

Die Infix-Notation bringt bei Spracherweiterungen jedoch große Nachteile. Zunächst muß der Operator exakt zwei Operanden (Argumente oder Parameter) haben – auf jeder Seite einen. Werden mehrere Operanden gebraucht, muß eine funktionelle Schreibweise wie FNf(a, b, c, d, e) verwandt werden. Weiterhin sind Ausdrücke wie 2+3*5 nicht eindeutig: Wird hier das + zuerst ausgeführt, oder das *? Das Problem läßt sich nur durch zusätzliche Regeln lösen, die beispielsweise besagen, daß * eine höhere Priorität hat als +. Wenn Sie bei der Infix-Notation neue Operatoren einführen wollen, müssen Sie daher auch die Regeln erweitern.

FORTH bietet hier eine Lösung, die den Fähigkeiten des Computers am besten entspricht: Bei allen Operatoren oder Funktionen (also allen Wörtern) werden zuerst die Operanden geschrieben und dann das Wort (beispielsweise 2 3 +). Die Schreibweise funktioniert wie ein Kochrezept: Zuerst holen Sie alle Zutaten, und dann verarbeiten Sie diese. Die technische Bezeichnung dieser Methode lautet „Umgekehrte Polnische Notation" (UPN). In der Normalen Polnischen Notation steht der Operator vor den Operanden. LOGO verwendet diese Schreibweise bei neuen Funktionen.

Nehmen wir an, Sie haben einen Ausdruck mit Infix-Notation wie 2*3+8/4. Da nach den üblichen Prioritätsregeln das + zuletzt ausgeführt wird, ist das Endergebnis die Summe von 2*3 und 8/4. Für die UPN-Schreibweise wird

der Ausdruck umgeformt:

(2*3) (8/4) +

Hier haben jedoch 2*3 und 8/4 noch die Infix-Notation. Die zweite Umformung ergibt daher:

(2 3 *) (8 4 /) +

Die UPN braucht keine Klammern, da sie in jedem Fall eindeutig ist. FORTH setzt Klammern außerdem nur für Kommentare ein. Hier das endgültige UPN-Format von 2*3+8/4:

2 3 * 8 4 / +

Bedenken Sie, daß in FORTH die Leerzeichen unbedingt notwendig sind.

Die Umgekehrte Polnische Notation kann also die beiden Probleme der Infix-Notation lösen: Bei der UPN sind pro Operator nicht nur zwei Operanden möglich, sondern mehrere (gefolgt von einem Operator). Beispielsweise hat in FORTH der Operator */ drei Operanden: Er multipliziert die ersten beiden und dividiert das Ergebnis durch den dritten Operanden.

Das zweite Problem der Infix-Notation sind die Prioritätsregeln und Klammern, durch die die Reihenfolge der Einzelberechnungen festgelegt wird. Die UPN kommt ohne Regeln und Klammern aus – sie teilt dem Computer exakt mit, wie das Ergebnis berechnet wird.

Ergebnis im Speicher

Sehen wir uns an dem einfachen Beispiel 2 3 + einmal an, wie die UPN ausgeführt wird. FORTH trifft zuerst auf die 2 und speichert sie. Danach findet es die 3, die ebenfalls gespeichert wird. Bei + nimmt der Computer an, daß er sich bereits zwei Zahlen „gemerkt" hat. Er findet sie, addiert sie und legt das Ergebnis wieder im Speicher ab. Dort steht es für die weitere Bearbeitung zur Verfügung.

In der letzten Folge erwähnten wir kurz, daß FORTH sich an Zahlen „erinnert", indem es diese auf den Stapel schiebt (nach der LIFO-Methode – Last In First Out). Dieses Konzept ist Maschinencodeprogrammierern sicherlich vertraut, doch kann es für Anfänger schon recht ungewohnt sein. Wir geben daher hier ein weiteres Beispiel. Stellen Sie sich einen Stapel Spielkarten vor. Um eine Nummer zu „speichern", schreiben Sie sie auf eine neue Karte, die Sie oben auf den Stapel legen. Für das + nehmen Sie die beiden obersten Karten herunter und addieren die beiden darauf notierten Zahlen, verändern jedoch nicht den Rest des Stapels. Der Kasten zeigt die Entwicklung des Stapels bei der Bearbeitung von 2 –3 * 8 4 / + ((2 * –3) + 8 / 4)). Beachten Sie, daß in Schritt 6 die Karte –6 von der Teilung 8 / 4 nicht berührt wird.

Zwar zeigt dieser Ablauf, wie der Stapel mit Ganzzahlen funktioniert, in FORTH werden Variablen jedoch als Adressen und nicht als Ganzzahlen behandelt. Die Adressen werden

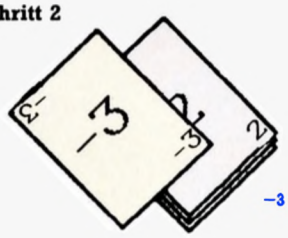
Von Oben nach Unten

Hier ein weiteres Beispiel, wie bei der Ausführung von 2 –3 * 8 4 / + (Infix Notation: 2* –3 + 8/4) Zahlen auf den Stapel „geschoben" und „heruntergezogen" werden.

Schritt 1

Die Zahl 2 wird zuoberst auf den Stapel gelegt...

Schritt 2

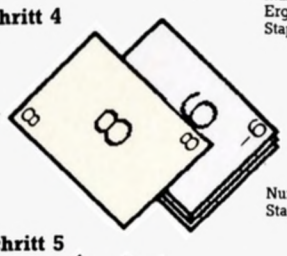


–3 wird auf die 2 gelegt

Schritt 3

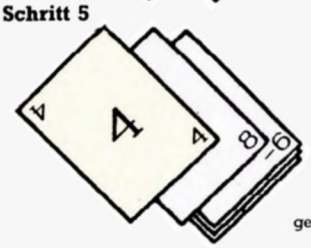
Das Wort * zieht die beiden Zahlen vom Stapel herunter, multipliziert sie und legt das Ergebnis wieder auf den Stapel.

Schritt 4



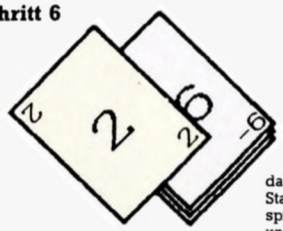
Nun wird 8 zuoberst auf den Stapel gelegt

Schritt 5



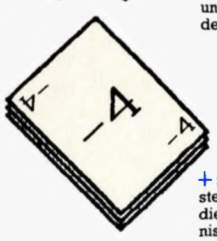
gefolgt von 4

Schritt 6



das / nimmt zwei Zahlen vom Stapel (–6 wird nicht angesprochen), verarbeitet sie und legt das Ergebnis wieder zuoberst auf den Stapel.

Schritt 7



+ zieht nun die beiden obersten Zahlen vom Stapel, addiert sie und legt das Ergebnis wieder auf den Stapel.

eingesetzt, um (mit @, dem Befehl für „holen") die Variablenwerte anzugeben oder (mit !, dem Befehl für „speichern"), um geänderte Werte auf die Adressen zu schreiben.

Unser Beispiel zeigt deutlich, wie exakt die Umgekehrte Polnische Notation arbeitet. FORTH unterhält eine eigene Version eines „Kartenstapels", so daß der Computer bei jedem Wort die entsprechenden Abläufe ausführen kann, ohne sich darum kümmern zu müssen, was zuvor oder danach passiert.

Hier ein weiteres Beispiel für den Einsatz des Stapels: Das Wort 2* ersetzt den letzten Wert des Stapels durch den doppelten Wert dieser Zahl. Es ist folgendermaßen definiert:

```
:2* (n —— n*2)
2 *
;
```

Beachten Sie, daß Klammern nur Kommentare umschließen. (Das Symbol —— wird in dem nebenstehenden Kasten erklärt.) Hier ein Beispiel für das neue Wort 2*:

```
19 2*.
```

zeigt als Ergebnis 38. Das Symbol . ist das FORTH-Wort für PRINT. Es nimmt den obersten Wert des Stapels und zeigt ihn auf dem Bildschirm an. Das *-Zeichen in der Definition von 2* erwartet mindestens zwei Zahlen. Die zweite Zahl ist dabei die in der Definition enthaltene 2, während sich die erste (in unserem Beispiel 19) bereits auf dem Stapel befinden muß, wenn 2* eingesetzt wird.

Stapelvorgänge dieser Art haben den Vorteil, daß sie mehr als ein Ergebnis erzeugen können. So beläßt das Wort /MOD beispielsweise zwei Werte auf dem Stapel: den „Quotienten" (das Ergebnis) und den „Rest" einer Division. Mit —— läßt sich dieser Vorgang aber auch folgendermaßen ausdrücken:

```
m, n —— Rest, Quotient von m/n
```

Bei der Infix-Notation wäre dieser Vorgang nicht möglich, bei der UPN ist er normal.

Die direkte Programmierung ist zwar eine der beeindruckendsten Eigenschaften von FORTH, kann aber auch Probleme verursachen. So stehen die Zahlen des Stapels zuweilen nicht in der richtigen Reihenfolge und müssen umorganisiert werden. Der nebenstehende Kasten enthält einige Standardwörter, die für diesen Zweck eingesetzt werden. Das nachfolgend definierte Wort */ ist zwar nicht so gut wie die Standarddefinition (die auch die richtige Antwort bietet, selbst wenn das Ergebnis zu groß für den Stapel ist), zeigt aber, wie ROT und SWAP funktionieren:

```
: */ (x,y,z——x*y/z)
  ROT ROT *
  SWAP /
;
```

Hier der Ablauf am Beispiel von 4 3 6 */:

Vorgang */	Stapel (oben → unten)
Am Anfang von */	4,3,6
ROT	3,6,4
ROT	6,4,3
*	6,12
SWAP	12,6
/	2
	Leer (2 wird dargestellt)

Normalerweise werden die Befehle zur Stapelmanipulation jedoch nur im Inneren von Wortdefinitionen eingesetzt.

Vorsicht

Sie müssen sorgfältig darauf achten, genau die Anzahl an Werten auf den Stapel zu schieben, die das Wort benötigt. Wenn Sie zu wenig anlegen, setzt das Programm selbständig Werte ein. Wurden zu viele eingesetzt, dann stimmt das Ergebnis ebenfalls nicht.

Für die genaue Wort-Definition können Sie die Schreibweise "——" einsetzen. Dabei wird die Liste der Werte, die ein Wort braucht, durch "——" von den Werten getrennt, die es auf dem Stapel zurückläßt. Hier einige Beispiele:

Wort:	Auswirkung:
+	m,n——m+n
.	m ——
*/	x, y, z——x*y/z

Die Schreibweise "——" ist kein Bestandteil von FORTH, sondern soll nur die Schreibweise übersichtlicher machen. Bei der Verwendung von Klammern sollten Sie beachten, daß nach der ersten Klammer ein Leerzeichen stehen muß. Die Klammer ist ein Wort, das besagt: Ignoriere von hier an alles, bis eine geschlossene Klammer auftaucht.

Abfrage des Stapels

Ein sehr praktisches Wort ist .S, das den Stapelinhalt anzeigt. Es ist im Standard-FORTH zwar nicht enthalten, in FORTH-83 aber definiert als:

```
: .S (——)
     (zeigt den Stapel von unten nach oben
       an)
     0 DEPTH 1 — DO
       I PICK.
     —1 + LOOP
;
```

In FORTH-79 funktioniert PICK anders. Die dritte Zeile muß dann lauten

```
  1 DEPTH DO
```

DEPTH (—— Stapeltiefe) informiert Sie darüber, wie viele Zahlen auf dem Stapel abgelegt waren, bevor Sie DEPTH aufgerufen haben. Da FigFORTH jedoch weder über DEPTH noch über PICK verfügt, funktionieren diese Definitionen dort natürlich nicht.

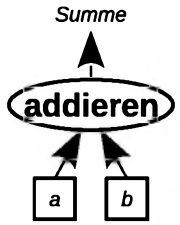
Quelle: *ComputerKurs*-Magazin, Heft 56, Seite 1556 – 1558, Jahrgang 1984/1985

1.12. Abschließende Betrachtungen zu Termbäumen

1.12.1. Termbäume en passant

Gleich zu Anfang dieser Abhandlung lernen Sie Termbäume (Rechenbäume) kennen. Sie erfuhren, daß diese eine wenig bekannte Strukturschreibweise der Mathematik sind. Mit ihnen können einfach Rechenausdrücke (Terme) abgebildet werden, und ihre Anschaulichkeit kann ein willkommener Vorteil sein.

Ungeachtet dessen, ob eine Addition mittels eines Pluszeichens ($+$, z.B. $a + b$) oder durch eine Additionsfunktion ($add()$, z.B. $add(a, b)$) niedergeschrienen ist, der Termbaum sieht für beide Varianten gleich aus:



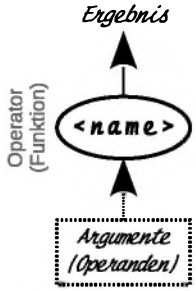
Für ein leichteres Verständnis darf man im Termbaum natürlich statt des Wortes „ ***addieren*** “ auch das Pluszeichen („ + “) bzw. das Wort „ ***add*** “ eintragen. Dadurch ändert sich der Termbaum, weder mathematisch, noch in seiner Struktur. Die Wahl, welche dieser drei Möglichkeiten Sie in den Termbaum eintragen, sollte so getroffen werden, daß das rasche Verstehen des Termbaums möglichst einfach bleibt.

Rein praktisch ermöglichen Termbäume ein synonymes Verstehen der Begriffe *Operator* und *Funktion*. Die mathematische Wissenschaft weiß in der Verwendung dieser beiden Begriffe fein zu unterscheiden, betrachten wir deren Nutzung aber via Termbäume, so erfüllt jede „*Operative-Einheit*“ in einem Termbaum genau einen Zweck: nämlich die Durchführung der zugewiesenen Berechnung.

Diese „Operative-Einheiten“ sind jene Teile eines Termbaumes, die wir in dieser Abhandlung durch Kreise (oder auch Ellipsen) dargestellt haben. Aus rein praktischen Gründen, und um uns den sprachlichen Umgang mit Termbäumen zu vereinfachen, sehen wir jede dieser „Operative-Einheiten“ als einen **Operator** an und bezeichnen diese auch so.

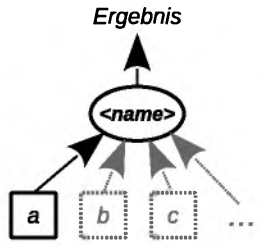
Ob wir nun zwei Zahlen addieren, multiplizieren, einen Logarithmus oder einen Sinus berechnen, oder ob wir die Wurzel aus einer Zahl ziehen aus unserer vereinfachenden Termbaumsicht sehen wir jede diese Berechnung als „Operative Einheit“ und somit als **Operator** an. Entsprechend nehmen wir eine weitere Vereinfachung vor und nennen alles, was man einer solchen „Operative Einheit“ (**Operator**) als Eingang zuführt **Operand**.

Die mathematische Wissenschaft trifft auch hierbei feinsinnige Unterscheidungen, wenn dort etwa von Argumenten, Parametern und Operanden die Rede ist, doch unsere vereinfachende Termbaumsicht läßt uns diese als Operanden bezeichnen. Wir sehen alles, was einem Operator als Eingang (Input) zugeführt wird, in dieser Abhandlung als Operand an. *Sind die hierbei vermittelten Inhalte verstanden, dürfen Sie sich gerne mit den feinsinnigen Betrachtungen der mathematischen Wissenschaft bezüglich Funktionen, Operatoren, Argumente, Parameter und Operanden vertraut machen.*

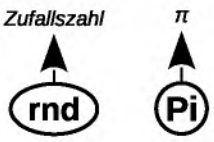


Grundsätzlich geben wir jedem Operator einen Namen. Das kann

- ein Wort / Stichwort sein, das dessen Wirkungsweise verständlich bezeichnet;
- ein Symbol / Zeichen der Mathematik, das der zugewiesenen Operation entspricht;
- eine Kurzform der Mathematik (z.B. „ *sin* “ für Sinus, „ *cos* “ für Cosinus oder „ *log* “ für Logarithmus);
- eine Abkürzung für Berechnungen, die in Wissenschaften, Berufsfeldern oder im Alltag genutzt werden (z.B. „ *USt.* “ für eine Umsatzsteuer-Berechnung).



Ferner legen Rechenvorschriften (Rechenregeln) fest, welche und wieviele Operanden einem Operator eingangs zugeführt werden. In der Regel sind dies ein oder zwei Operanden, manchmal auch mehrere; doch es gibt auch Operatoren, die ohne einen Operanden auskommen:



- Operatoren, die eine Zufallszahl liefern (z.B. „ *rnd* “ für eine *random-number*-Operation);
- Operatoren, die eine festgelegte Zahl liefern (z.B. die Zahl „Pi“ π).

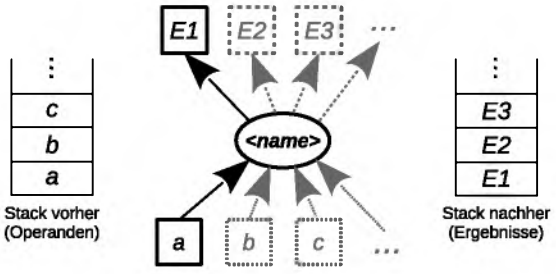
In Programmiersprachen erkennen Programmierer anhand der gestellten Aufgabe, welche und wieviele Operatoren einem Operator (in Programmiersprachen auch *Routine* genannt) eingangs zuzuführen sind. Grundsätzlich sollten ausschließlich Operanden zugeführt werden, die für die jeweilige Operation notwendig sind; alles was für die jeweilige Operation nicht gebraucht wird, ist als unnötiger Ballast anzusehen und sollte nicht zugeführt werden.

Beispiel: Sollten Sie später einmal eine für ein Computerprogramm eine Umsatzsteuerberechnung programmieren, dann fließen ein Warenwert und ein Steuersatz in diese Berechnung ein und werden der Operation als Operanden zugeführt.

Für diese Umsatzsteuerberechnung dürfe jedoch nicht von Bedeutung sein, ob Sie gerade Socken in Ihrer heimischen Waschmaschine waschen und wen ja, wieviele. Diese Daten werden daher für Ihre Umsatzsteuerberechnung nicht erhoben und dieser auch nicht als weitere Operanden zugeführt.

Sollten Sie jedoch einmal ein eigens Steuerungsprogramm für Ihre heimische Waschmaschine entwickeln (d.h. programmieren), dann könnten solche Daten durchaus an gewissen Stellen Beachtung finden.

1.12.2. Etwas, das Infix-Notation nicht kann



Infix-Terme liefern genau ein Ergebnis, und zwar immer. Für Postfix-Terme gilt diese Begrenzung nicht. Sie können eine beliege Anzahl an Ergebnissen zurück geben, die mit Abschluß einer Operation auf dem Stack abgelegt sind. Und von nachfolgenden Operanden aufgegriffen werden können. Und dies ist bei Postfix ein übliches Vorgehen.

Bei Postfix können also Operatoren genutzt werden, die in der Infix-Welt nicht existieren.

Ein solcher Postfix-Operator ist beispielsweise ***/mod*** . Dieser Operator erwartet bei seiner Ausführung zwei Operanden *a* und *b* auf dem Stack. Mit seiner Ausführung dividiert er *a* durch *b* und legt anschließend das ganzzahlige Ergebnis *E1* und den Divisionsrest *E2* auf den Stack ab. Diese können danach von weiteren Operatoren genutzt werden.

Auch Präfix (NPN) vermag mit solches zu handhaben. Die Frage ist aber, ob das den jeweiligen Anwendern von Präfix bewußt ist. Anwendern von Postfix (UPN) ist dieses in der Regel sehr bewußt.

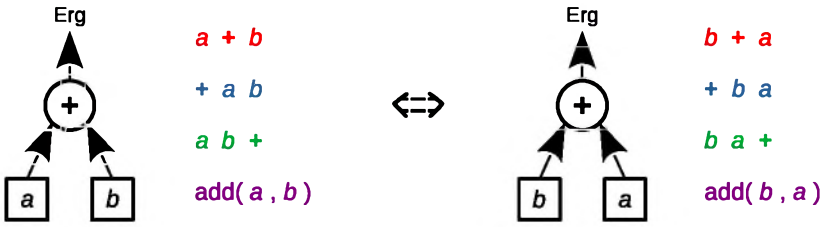
1.13. Grundlegende mathematische Operatoren

Zum Abschluß erhalten Sie hier eine Übersicht grundlegender mathematischer Operatoren als Termbaum samt zugehöriger Infix-Notation (rot), Präfix-Notation / NPN (blau), Postfix-Notation / UPN (grün) und einer Funktionenschreibweise, die an Tabellenkalkulationsprogrammen und Programmiersprachen erinnern soll.

1.13.1. Operatoren mit zwei Operanden

Addition

Summand plus Summand gleich Summe

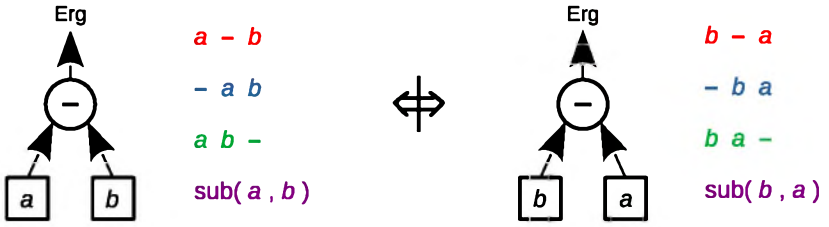


Erg	:	*Ergebnis (Summe)*
a	:	*Operand (Summand)*
b	:	*Operand (Summand)*

Bei der *Addition* sind der linke der rechte Term äquivalent; somit dürfen die Operanden untereinander vertauscht werden, ohne daß dies Einfluß auf das Ergebnis hat.

Subtraktion

Minuend minus Subtrahend gleich Differenz

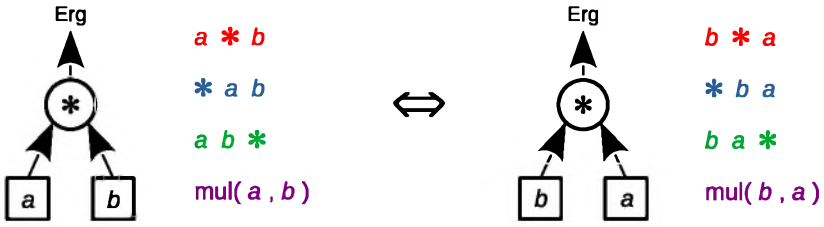


Erg	:	*Ergebnis (Differenz)*
a	:	*Operand (Minuend)*
b	:	*Operand (Subtrahend)*

Bei der *Subtraktion* sind der linke der rechte Term nicht äquivalent; somit dürfen die Operanden nicht untereinander vertauscht werden, da dies prinzipiell Einfluß auf das Ergebnis hat.

Multiplikation

Faktor mal Faktor gleich Produkt

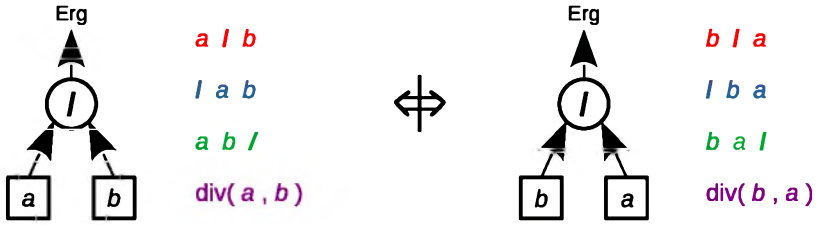


Erg	:	*Ergebnis (Produkt)*
a	:	*Operand (Faktor)*
b	:	*Operand (Faktor)*

Bei der *Multiplikation* sind der linke der rechte Term äquivalent; somit dürfen die Operanden untereinander vertauscht werden, ohne daß dies Einfluß auf das Ergebnis hat.

Division

Dividend geteilt durch Divisor gleich Quotient



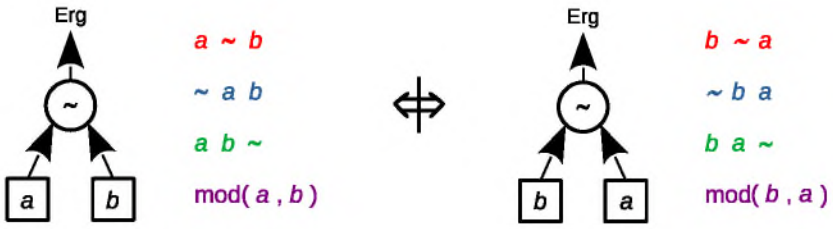
Erg	:	*Ergebnis (Quotient)*
a	:	*Operand (Dividend)*
b	:	*Operand (Divisor)*

Bei der *Division* sind der linke der rechte Term <u>nicht</u> äquivalent; somit dürfen die Operanden <u>nicht</u> untereinander vertauscht werden, da dies prinzipiell Einfluß auf das Ergebnis hat.

Division mit Rest

(Modulo-Division)

Dividend rest-geteilt durch Divisor gleich Rest
(Dividend modulo durch Divisor gleich Rest)

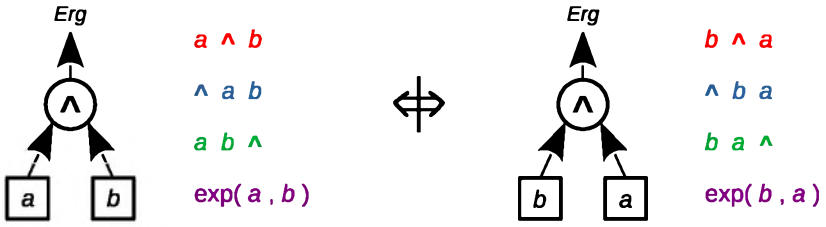


Erg	:	*Ergebnis (Rest)*
a	:	*Operand (Dividend)*
b	:	*Operand (Divisor)*

Bei der *Division mit Rest* sind der linke der rechte Term <u>nicht</u> äquivalent; somit dürfen die Operanden <u>nicht</u> untereinander vertauscht werden, da dies prinzipiell Einfluß auf das Ergebnis hat.

Potenzieren
(Exponentiation)

Basis hoch Exponent gleich Potenz[wert]



Erg	:	*Ergebnis (Potenz)*
a	:	*Operand (Basis)*
b	:	*Operand (Exponent)*

Bei der *Potenz* sind der linke der rechte Term nicht äquivalent; somit dürfen die Operanden nicht untereinander vertauscht werden, da dies prinzipiell Einfluß auf das Ergebnis hat.

Wurzelziehen
(Radizieren)

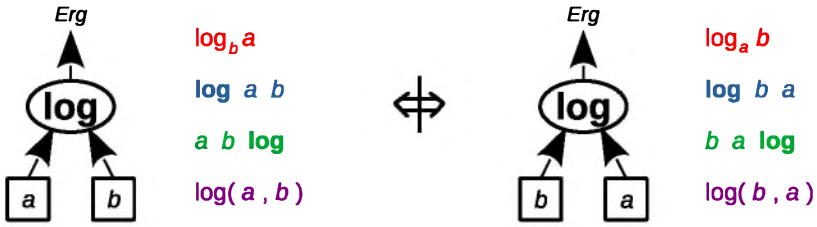
Radikant radiziert durch Wurzelexponent gleich Radikal
(Wurzel von Radikant durch Wurzelexponent gleich Wurzelwert)

Erg	:	*Ergebnis (Radikal)*
a	:	*Operand (Radikant)*
b	:	*Operand (Wurzelexponent)*

Bei der *Wurzel* sind der linke der rechte Term nicht äquivalent; somit dürfen die Operanden nicht untereinander vertauscht werden, da dies prinzipiell Einfluß auf das Ergebnis hat.

Logarithmus

Logarithmus von Numerus zur Basis gleich Logarithmuswert



Erg	:	*Ergebnis (Logarithmus[wert])*
a	:	*Operand (Numerus)*
b	:	*Operand (Basis)*

Beim *Logarithmus* sind der linke der rechte Term nicht äquivalent; somit dürfen die Operanden nicht untereinander vertauscht werden, da dies prinzipiell Einfluß auf das Ergebnis hat.

1.13.2. Operatoren mit einem Operand

Quadrat

a^2

^2 *a*

a ^2

sqr(*a*)

Erg : *Ergebnis*
a : *Operand*

Liefert die Quadratzahl zu *a*.

Quadratwurzel

$\sqrt{a}$

√ *a*

a √

sqrt(*a*)

Erg : *Ergebnis*
a : *Operand*

Liefert die Quadratwurzel von *a*.

Fakultät

n!

! *n*

n !

fac(*n*)

Erg : *Ergebnis*
a : *Operand*

Liefert die Fakultät von *a*.

Betrag
(Absolutwert)

|*a*|

abs *a*

a abs

abs(*a*)

Erg : *Ergebnis*
a : *Operand*

Liefert den die Fakultät von *a*.

dekadischer Logarithmus
(Zahlenbasis ist *10*)

lg *a*

lg *a*

a lg

lg(*a*)

Erg : *Ergebnis*
a : *Operand*

Liefert den dekadischen Logarithmus[wert] von *a*.

natürlicher Logarithmus
(Zahlenbasis ist die Eulersche Zahl e)

ln *a*

ln *a*

a ln

ln(*a*)

Erg : *Ergebnis*
a : *Operand*

Liefert den natürlichen Logarithmus[wert] von *a*.

Sinus

sin *a*

sin *a*

a sin

sin(*a*)

Erg : *Ergebnis*
a : *Operand*

Liefert den Sinus[wert] von *a*.

Cosinus

cos *a*

cos *a*

a cos

cos(*a*)

Erg : *Ergebnis*
a : *Operand*

Liefert den Cosinus[wert] von *a*.

Tangens

tan *a*

tan *a*

a tan

tan(*a*)

Erg : *Ergebnis*
a : *Operand*

Liefert den Tangens[wert] von *a*.

Cotangens

cot *a*

cot *a*

a cot

cot(*a*)

Erg : *Ergebnis*
a : *Operand*

Liefert den Cotangens[wert] von *a*.

1.13.3. Operatoren ohne Operand

Die Kreiszahl
Pi

π
π
π
pi()

Erg : *Ergebnis*

Liefert den Kreiszahl *π*.

Das ist eine unendliche Zahl, die mit der Ziffernfolge *3,14159265 ...* beginnt.

Die Eulersche Zahl
e

e
e
e
e()

Erg : *Ergebnis*

Liefert die Eulersche Zahl *e*.

Das ist eine unendliche Zahl, die mit der Ziffernfolge *2,71828182...* beginnt.

Zufallszahl
(random number)

rnd
rnd
rnd
rnd()

Erg : *Ergebnis*

Liefert eine zufällige Zahl (Zufallszahl).

Fließt eine solche Zufallszahl in ein Kette weiterer Operatoren ein, läßt sich am Ende der Wertebereich des Gesamtergebnisses geschickt eingrenzen.

Programmiersprachen stellen teil-weise einen derartigen Operator bereit.